

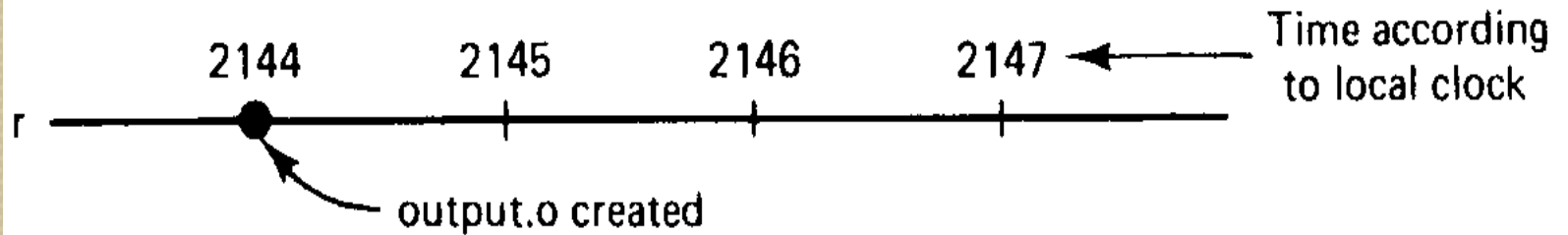
# **Synchronization in Distributed Systems**

# Why do we need synchronization?

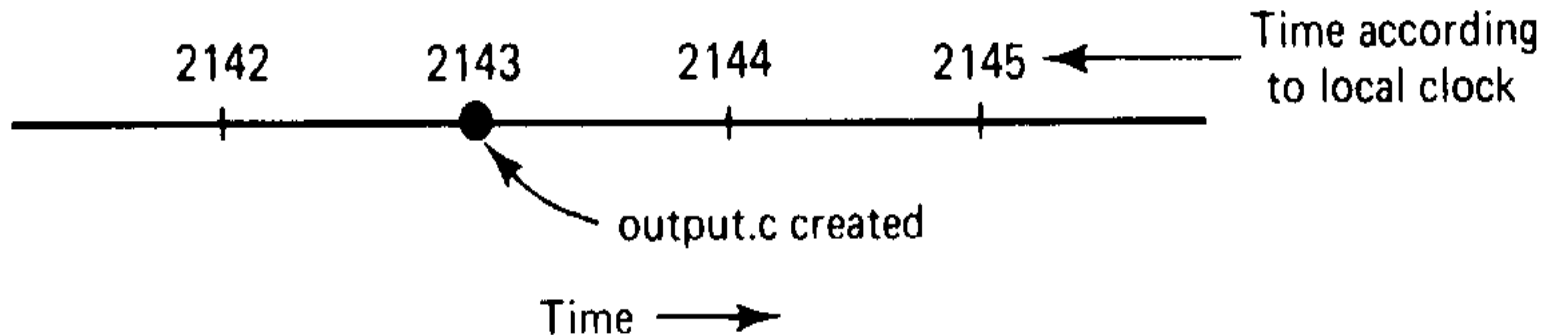
- As we have seen, clients and servers are communicating together to form a DS.
- File access, process scheduling and allocation, algorithms running in different machines, clients accessing a server needed to be scheduled in a queue,.. All these requires synchronization.
- They need to adjust their time, i.e. synchronize, together.

# Example of 2 different clients having different time

**Client  
A**



**Client  
B**



Since time is so basic to the way people think, and the effect of not having all the clocks synchronized can be so dramatic, as we have just seen, it is fitting that we begin our study of synchronization with the simple question: Is it possible to synchronize all the clocks in a distributed system?

- The Concept of Time
  - A standard time is a set of instants with a temporal precedence order

# Time as a Partial Order

- A linearly ordered structure of time is not always adequate for distributed systems
  - Captures dependence, not independence of distributed activities
- A partially ordered system of *vectors* forming a *lattice* structure is a natural representation of time in a distributed system
- Resembles Einstein-Minkowski's relativistic space-time

# Computer's Logical Time

- Every computer has a clock that keeps track of the time, it is better called a **timer**.
- It is constructed from quartz crystal which oscillates at specified frequency.
- It is connected to a counter and a holding register.
- Each oscillation decrements the counter by one.
- When the counter gets to zero, an interrupt is called which reloads the time from the holding register to the counter.
- Each interrupt call is named clock tick.

# Computer's Logical Time

- When a computer is booted initially, the user is asked to enter the time which is used as a start for the ticks and stored in memory.
- At every clock tick, the interrupt procedure adds one to the time stored in memory.
- That is how the **software clock** is kept up to date.
- As we have multiple CPUs in DS, so we have multiple clocks.
- The difference between their time is called **clock skew**.



# Logical Clock Synchronization



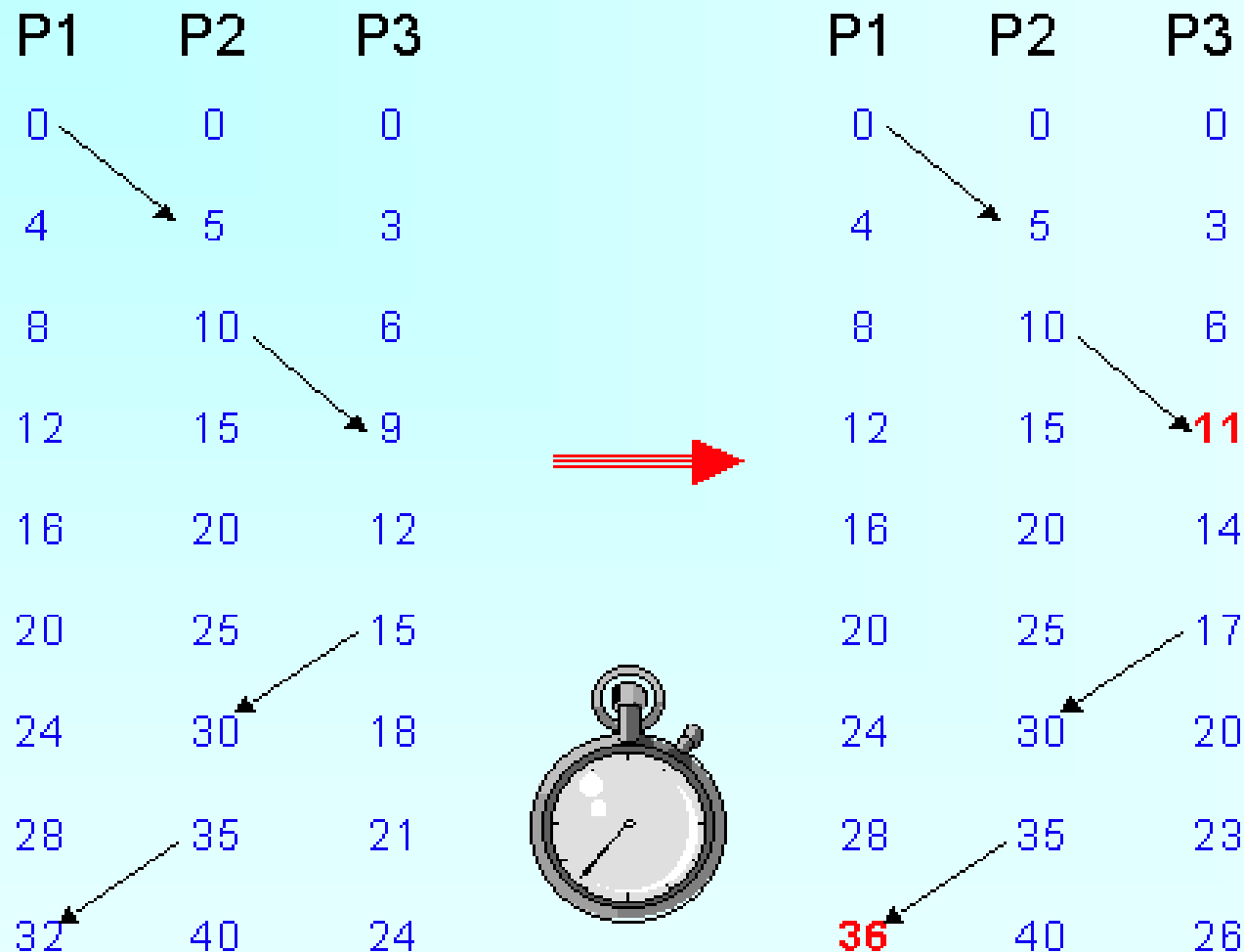
# Event Ordering

- Lamport (1978 – modified 1990) defined the “happens before” ( $<$ ) relation
  - If  $a$  and  $b$  are events in the same process, and  $a$  occurs before  $b$ , then  $a < b$ .
  - If  $a$  and  $b$  at different machines that are not communicating, then their order is not known.
  - If  $a$  is the time of a message being sent by one process, and  $b$  is the time of that message being received by another process, then  $a < b$  is true.
  - A message can not be received unless it is sent.

# Causal Ordering

- “Happens Before” also called **causal ordering**
- Possible to make a causality relation between 2 events if
  - They happen in the same process
  - There is a chain of messages between them
- “Happens Before” notion is not straightforward in distributed systems
  - No guarantees of synchronized clocks
  - Communication latency

# Lamport Logical Clock



**Fig. 3-2.** (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

# Logical Clocks

- Used to determine causality in distributed systems
- Time is represented by non-negative integers
- A logical Clock  $C$  is some abstract mechanism which assigns to any event  $e \in E$  the value  $C(e)$  of some time domain  $T$  such that certain conditions are met
  - $C: E \rightarrow T :: T$  is a partially ordered set :  
 $e < e' \rightarrow C(e) < C(e')$  holds
- Consequences of the clock condition [**Morgan 85**]:
  - If an event  $e$  occurs before event  $e'$  at some single process, then event  $e$  is assigned a logical time earlier than the logical time assigned to event  $e'$
  - For any message sent from one process to another, the logical time of the send event is always earlier than the logical time of the receive event

# Independence

- Two events  $e, e'$  are mutually independent (i.e.  $e \parallel e'$ ) if  $\sim(e < e') \wedge \sim(e' < e)$ 
  - Two events are independent if they have the same timestamp

# Problems with Total Ordering

- A linearly ordered structure of time is not always adequate for distributed systems
  - captures dependence of events
  - loses independence of events - artificially enforces an ordering for events that need not be ordered.
  - Mapping partial ordered events onto a linearly ordered set of integers it is *losing information*
    - Events which may happen simultaneously may get different timestamps as if they happen in some definite order.

# Time Manager Operations

- Logical Clocks
  - C.adjust(L,T)
    - adjust the local time displayed by clock C to T (can be gradually, immediate, per clock sync period)
  - C.read
    - returns the current value of clock C
- Timers
  - TP.set(T) - reset the timer to timeout in T units
- Messages
  - receive(m,l); broadcast(m); forward(m,l)



# Physical Clock Synchronization



# Physical Clocks

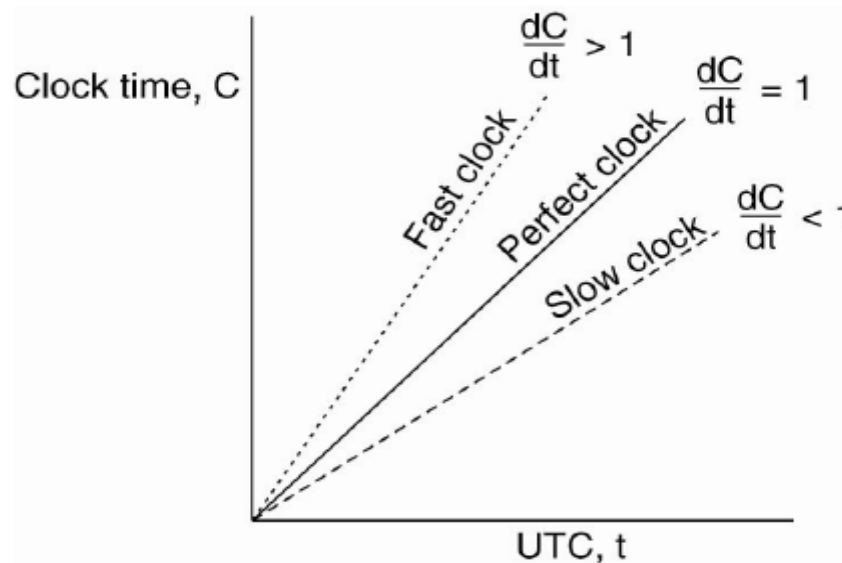
- How do we measure real time?
  - 17th century - Mechanical clocks based on astronomical measurements
    - Solar Day - Transit of the sun
    - Solar Seconds -  $\text{Solar Day} / (3600 \times 24)$
  - Problem (1940) - Rotation of the earth varies (gets slower)
  - Mean solar second - average over many days

# Atomic Clocks

- 1948
  - counting transitions of a crystal (Cesium 133) used as atomic clock
  - TAI - International Atomic Time
  - UTC (Universal Coordinated Time)
    - From time to time, we skip a solar second to stay in phase with the sun (30+ times since 1958)
    - UTC is broadcast by several sources (satellites...)

# Accuracy of Computer Clocks

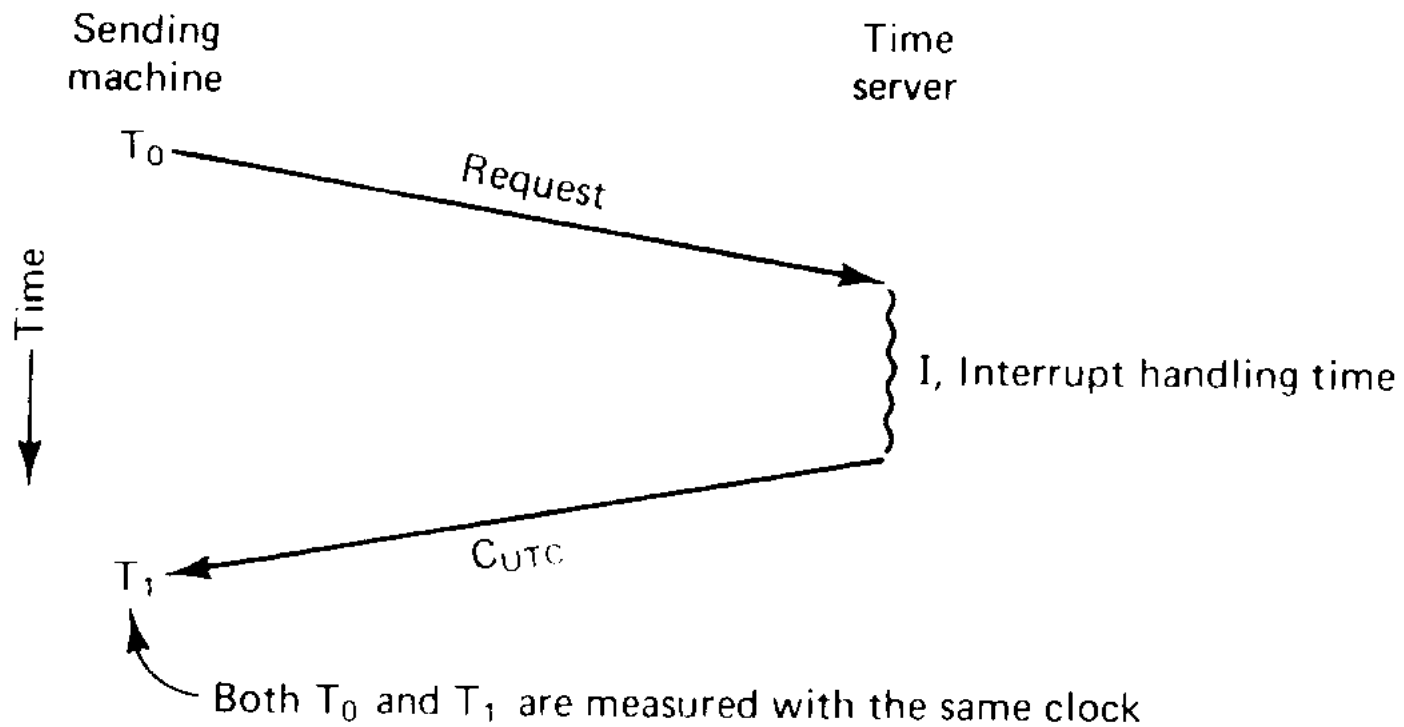
- Modern timer chips have a relative error of  $1/100,000$  - 0.86 seconds a day



- To maintain synchronized clocks
  - **Sol 1:** Use UTC source (time server) to obtain current notion of time
  - **Sol 2:** Use solutions without UTC.

# Cristian's (Time Server) Algorithm (1989) - Sol I

- Uses a *time server* to synchronize clocks (time server is passive)
  - Time server keeps the reference time (say UTC)
  - A client asks the time server for time, the server responds with its current time, and the client uses the received value *T* to set its clock
- But network round-trip time introduces errors...
  - Let **RTT (Round Trip Time) = response-received-time – request-sent-time** (measurable at client),
  - If we know (a)  $\min$  = minimum client-server one-way transmission time and (b) that the server timestamped the message at the last possible instant before sending it back



**Fig. 3-6.** Getting the current time from a time server.

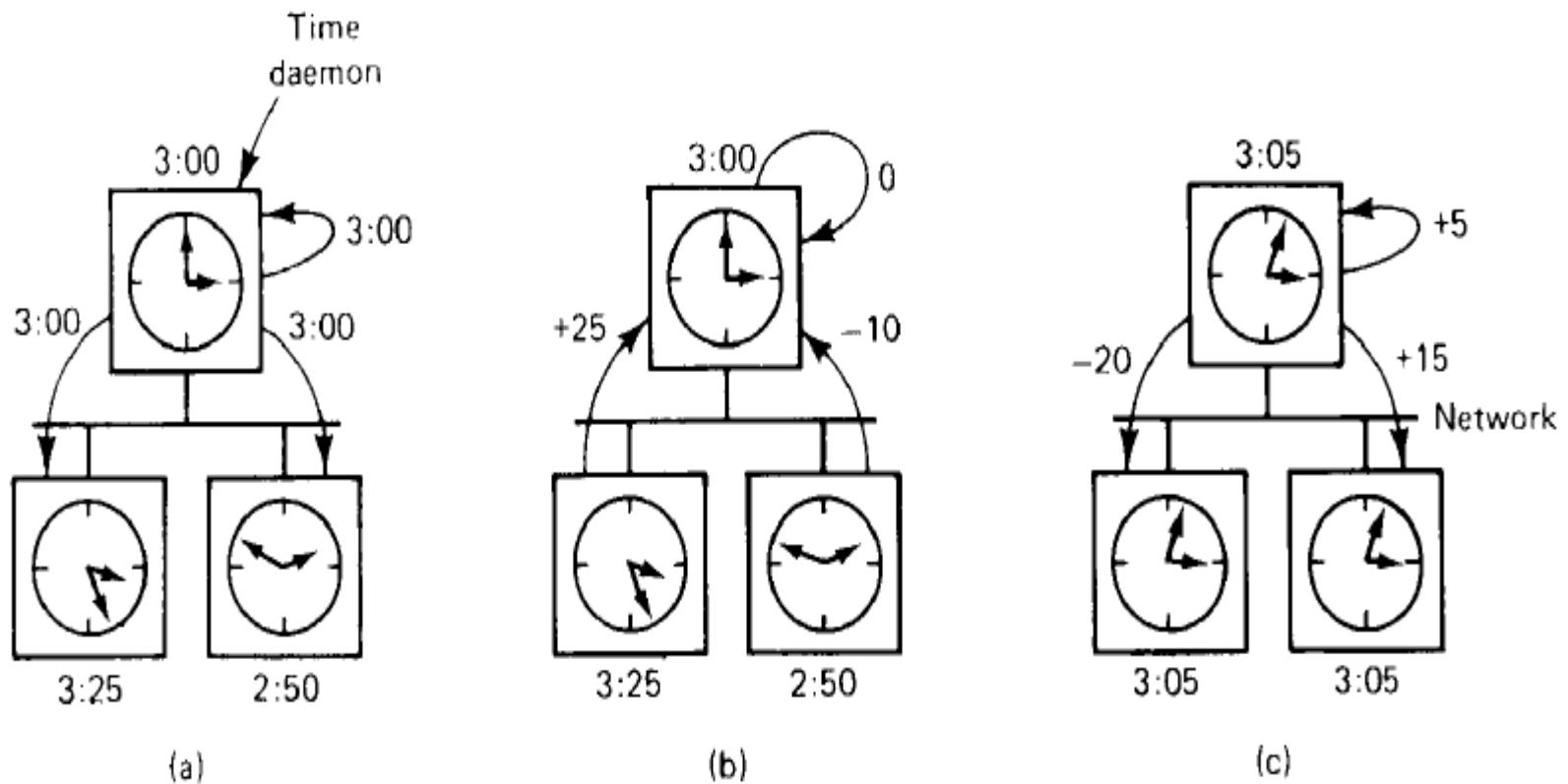
# Cristian's Algorithm

- ♣ Then, the actual time could be between  $[T + \min, T + \text{RTT} - \min]$
- ♣ Client sets its clock to halfway between  $T + \min$  and  $T + \text{RTT} - \min$  i.e., at  $T + \text{RTT}/2$ 
  - ☹ Expected (i.e., average) skew in client clock time =  $(\text{RTT}/2 - \min)$
- ♣ Can increase clock value, should never decrease it.
- ♣ Can adjust speed of clock too (either up or down is ok)
- ♣ Multiple requests to increase accuracy
  - ♣ For unusually long RTTs, repeat the time request
  - ♣ For non-uniform RTTs
    - ♣ Drop values beyond threshold; Use averages (or weighted average)

# Berkeley UNIX algorithm

## Sol 2

- It has one daemon or time server that does not have UTC (time server is active)
- Periodically, this daemon polls and asks all the machines for their time
- The machines respond with the difference in time to the daemon time.
- The daemon computes an average time and then broadcasts the effect of this average time to himself and the other machines.



**Fig. 3-7.** (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

- It tells the other machines to advance or to slow down their clock
- To be accurate, the time server clock must be set manually by the admin to UTC



# Decentralized Averaging Algorithm

## Sol 2

- The time is divided into intervals
- Periodically, at the beginning of each interval, each machine broadcasts its local time to all machines.
- Each of them calculates the average time by averaging all the received local times.
- Because their time is not accurate, every machine has a timer to collect the broadcasted times arriving in period  $S$ .

# Decentralized Averaging Algorithm

## Sol 2

- Other variations, is to discard the largest and the smallest time before taking the average.
- Also, they can put the round-trip time into consideration before the average.

# Clock Synchronization in DCE

- DCE's time model is actually in an interval
  - I.e. time in DCE is actually an interval
  - Comparing 2 times may yield 3 answers
    - $t_1 < t_2$
    - $t_2 < t_1$
    - not determined
  - Each machine is either a time server or a clerk
  - Periodically a clerk contacts all the time servers on its LAN
  - Based on their answers, it computes a new time and gradually converges to it.

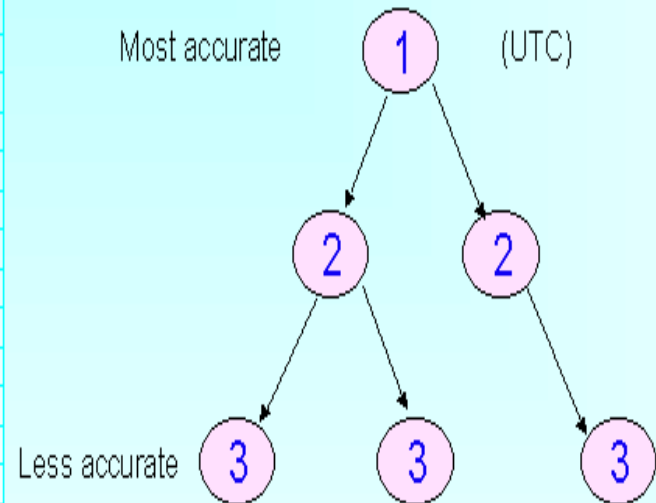
# Network Time Protocol (NTP)

- Most widely used physical clock synchronization protocol on the Internet (<http://www.ntp.org>)
- 10-20 million NTP servers and clients in the Internet
- Claimed Accuracy (Varies)
  - milliseconds on WANs, submilliseconds on LANs, submicroseconds using a precision timesource
  - Nanosecond NTP in progress

# NTP Design

- Hierarchical tree of time servers.
  - The primary server at the root synchronizes with the UTC.
  - The next level contains secondary servers, which act as a backup to the primary server.
  - At the lowest level is the synchronization subnet which has the clients.

## Hierarchy in NTP



# NTPs Offset Delay Estimation Method

- NTP performs several trials and chooses trial with minimum delay

Let  $a = T_1 - T_3$  and  $b = T_2 - T_4$ .

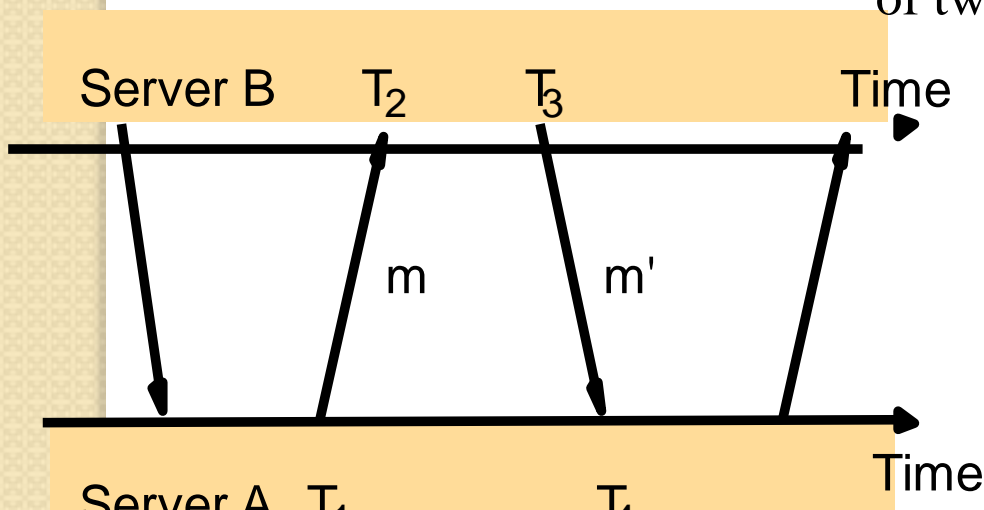
- If differential delay is small, the clock offset  $\Theta$  and roundtrip delay  $\delta$  of B relative to A at time  $T_4$  are approximately given by

$$\Theta = (a + b)/2, \delta = a - b$$

- A pair of servers in symmetric mode exchange pairs of timing messages.

- A store of data is then built up about the relationship between the two servers (pairs of offset and delay).

- Specifically, assume that each peer maintains pairs  $(O_i, D_i)$ , where  $O_i$  - measure of offset;  $D_i$  - transmission delay of two messages.





# Mutual Exclusion of Processes in Distributed Systems

# Mutual Exclusion in Distributed Systems

- When multiple processes are communicating through shared memory (called critical regions), the problem of reading or updating a shared data occurs.
- They must ensure that this data is not read/written by more than one process at the same time.
- They use what so called mutual exclusion to prevent inconsistent usage or updates to shared data.
- Two approaches
  - *Token*
  - *Permission*



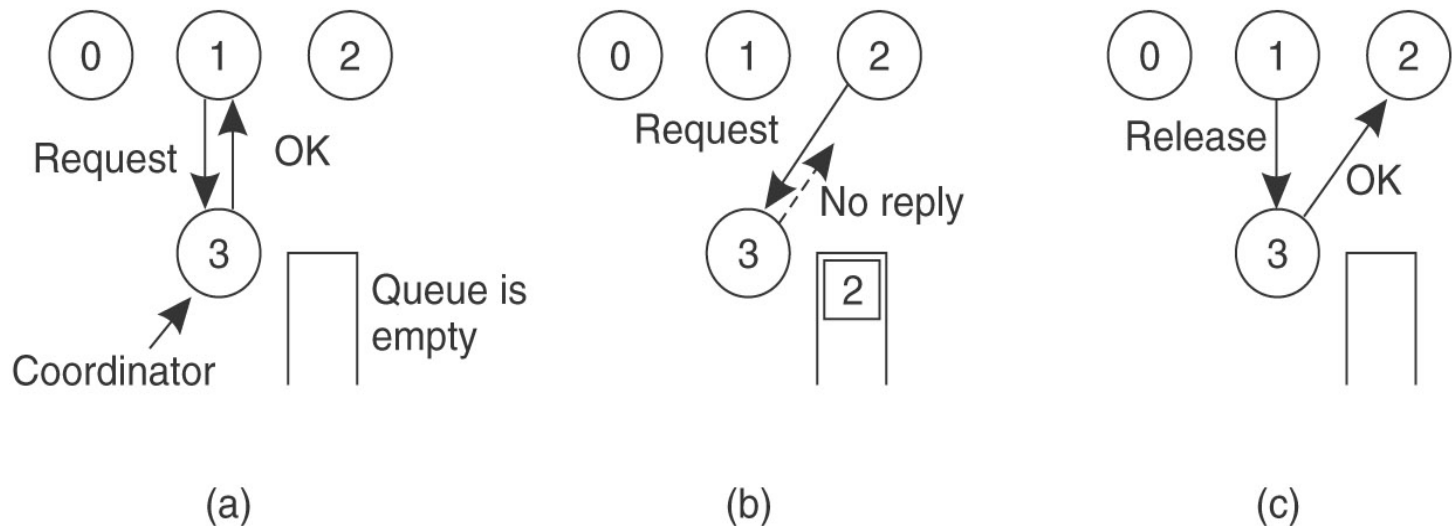
# Distributed Mutual Exclusion

- *Token-based solutions:* Based on passing a special message between the processes known as a *token*. There is only one token available and the process that has it can enter the critical section. It avoids starvation and deadlocks. But what if the token gets lost?
- *Permission-based approach:* A process wanting to enter the critical section first requires permission of other processes. Some ways of getting the permission:
  - Centralized algorithm
  - Decentralized algorithm
  - Distributed algorithm

# Centralized *Permission* Approach

- It simulates what happen in single processor system.
- One process is elected as *coordinator* for a resource
- Whenever a process wants to enter the critical region, it asks the *permission* from the coordinator.
- Possible responses
  - Okay; denied (ask again later); none (caller waits)

- If no other processes is using this region, this process is granted the permission.
- When the reply is received by the process, it starts using the region.



**Fig. 3-8.** (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

# Centralized *Permissions* (continued)

- Advantages
  - Mutual exclusion guaranteed by coordinator
  - “Fair” sharing possible without starvation
  - Simple to implement
- Disadvantages
  - Single point of failure (coordinator crashes)
  - Performance bottleneck

# Distributed version

- Each resource is replicated  $n$  times with one coordinator per replica.
- To access a resource we need to get a vote from a majority of the coordinators.
- If access is denied, the process is informed.
- Implemented using DHT (Distributed Hash Table) based system.
- Has the problem if many nodes are competing to get the critical section, no one is able to get enough votes.
- Possible to incorrectly give access because of coordinator crashing and missing information.

# Decentralized Permissions

- $n$  coordinators; ask all
  - E.g.,  $n$  replicas
- Must have agreement of  $m > n/2$
- Advantage
  - No single point of failure
- Disadvantage
  - Lots of messages
  - Really messy

# Distributed Permissions

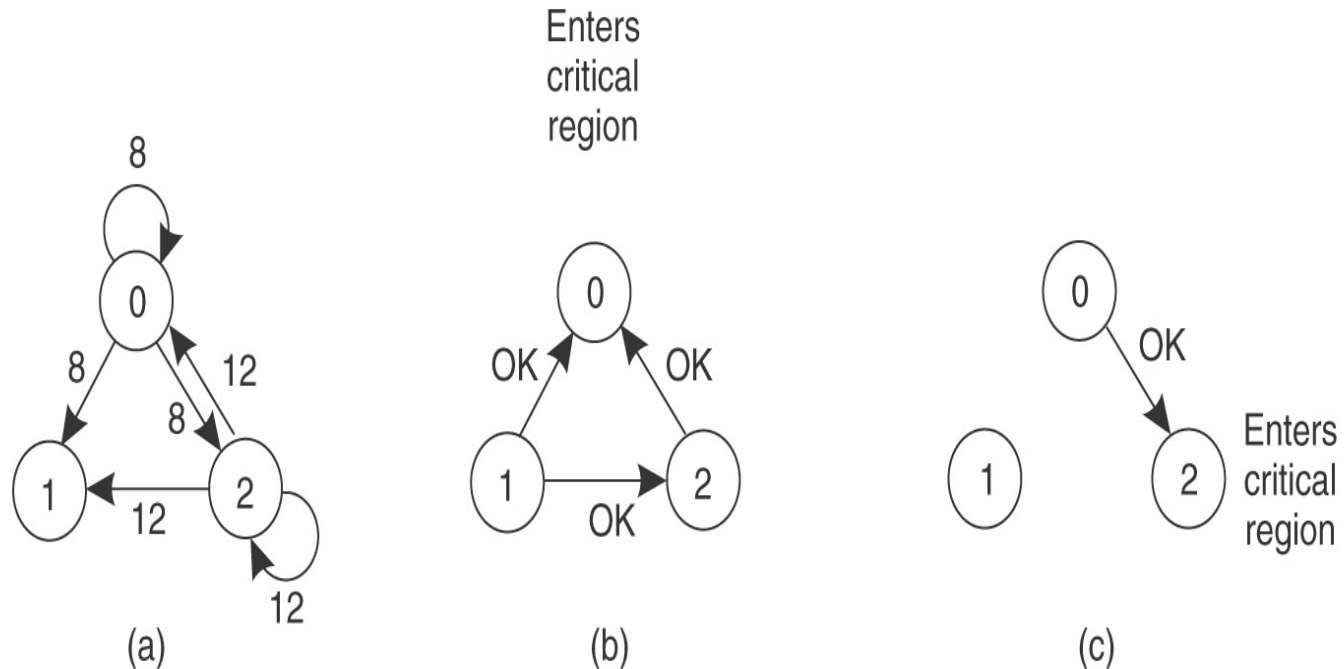
- Use Lamport's logical clocks
- Requestor sends reliable messages to *all* other processes (including self) containing the name of the resource, its process number and its current logical time.
  - Waits for *OK* replies from all other processes
- Replying process has 3 possibilities:
  1. If not interested in resource, reply OK
  2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request
  3. If interested, then reply OK if requestor is earlier than itself, otherwise not interested

# Distributed Permissions

3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. Sends an OK if incoming timestamp is lower.



# Distributed Permissions (Example)



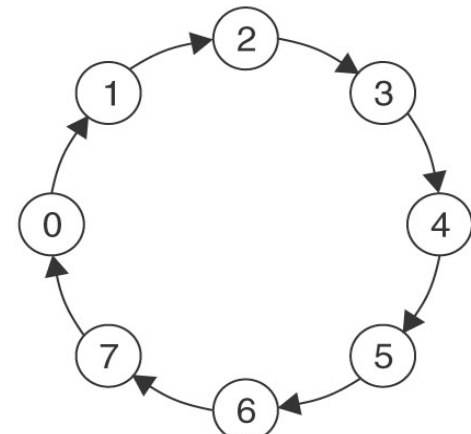
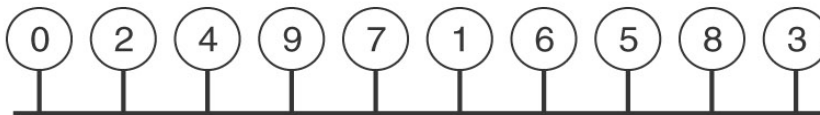
- Process 0 and Process 2 want resource. Process 1 is not interested.
- Process 1 replies OK because not interested
- Process 0 has lower time-stamp, thereby goes first

# Distributed Permissions (continued)

- Advantage
  - No central bottleneck
  - Fewer messages than Decentralized
- Disadvantage
  - $n$  points of failure
  - i.e., failure of one node to respond locks up system

# Token system

- Organize processes in logical *ring* (actually they are connected by a bus but the SW simulates a ring)
- Each process knows its successor
- Token is passed around ring
  - If process is interested in resource, it waits for token from its neighbor
  - Releases token when done



# *Token system* (continued)

- If node is dead or doesn't need the critical region, process skips over it
  - Passes token to successor of dead process
- Advantages
  - Fairness, no starvation, no conflict occurs
  - Recovery from crashes if token is not lost
- Disadvantage
  - If token is lost, it has to be regenerated.
  - Difficult to detect even if took time with the token not found
  - Crash of process holding token

# Next Time

- Election algorithms for synchronization
- Consistency and Replication

# Election algorithms for synchronization

- If a group of processes want to synchronize, they have to choose (i.e. elect) a coordinator to do so.
- There are several methods to make the elections which assumes that each process has a number distinguishing it.

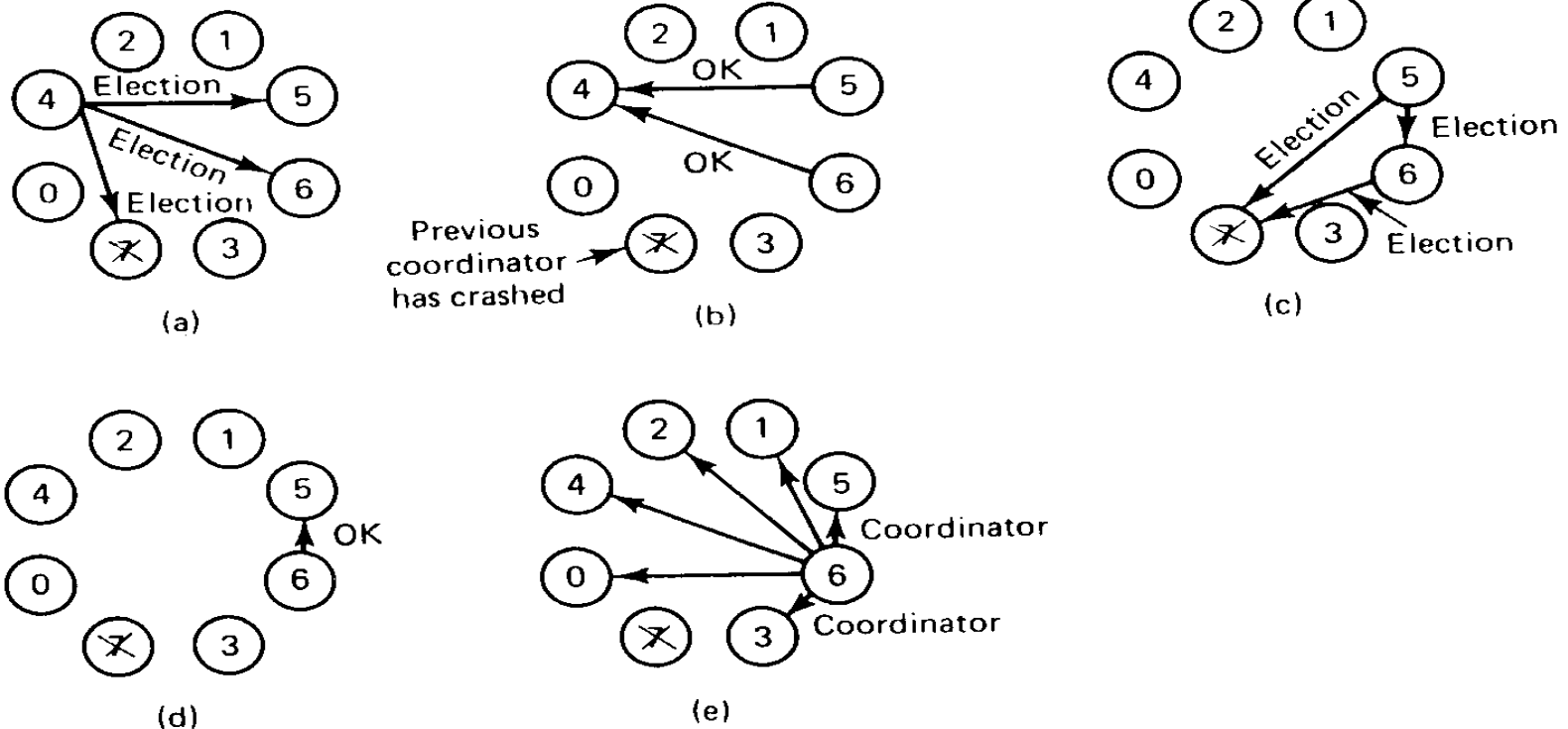
## I- Bully Algorithm (1982)

When a process notices that the coordinator is not responding to requests, it initiates an election as follows.

# I - Bully election algorithm

1.  $P$  sends an *ELECTION* message to all processes with higher numbers.
  2. If no one responds,  $P$  wins the election and becomes coordinator.
  3. If one of the higher-ups answers, it takes over.  $P$ 's job is done.
- If a process **A** gets an election message from one of its lower-numbered processes, it sends OK message to indicate it is alive and will take over the role of coordinator.
  - Then process **A** holds an election until all other processes give up but one which will be the coordinator.
  - It announces its victory by sending all processes a message telling them it is the coordinator

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

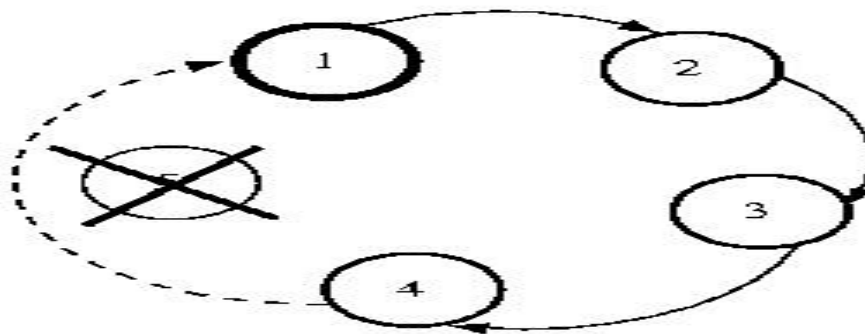
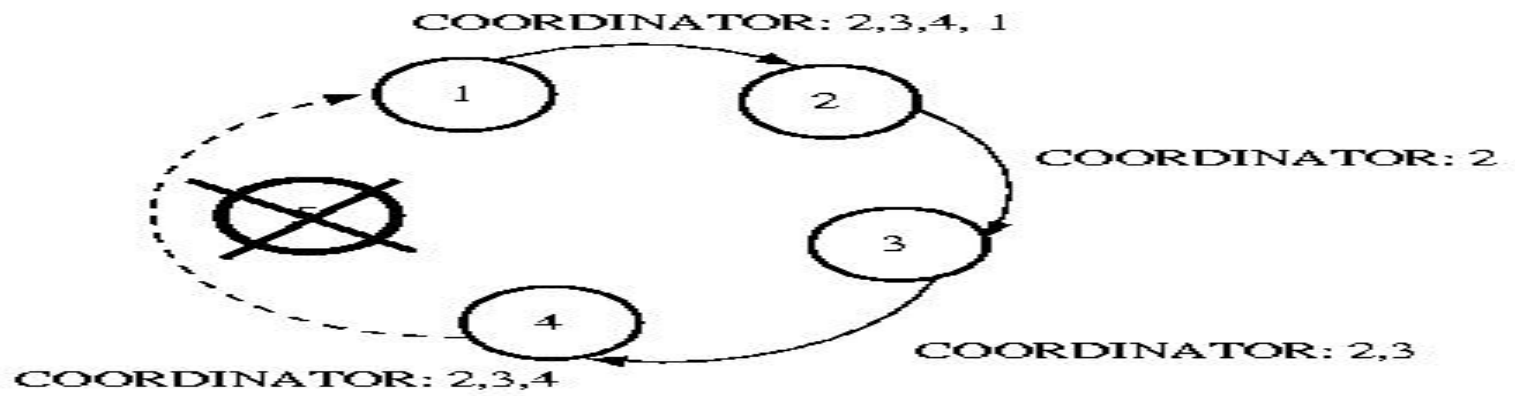
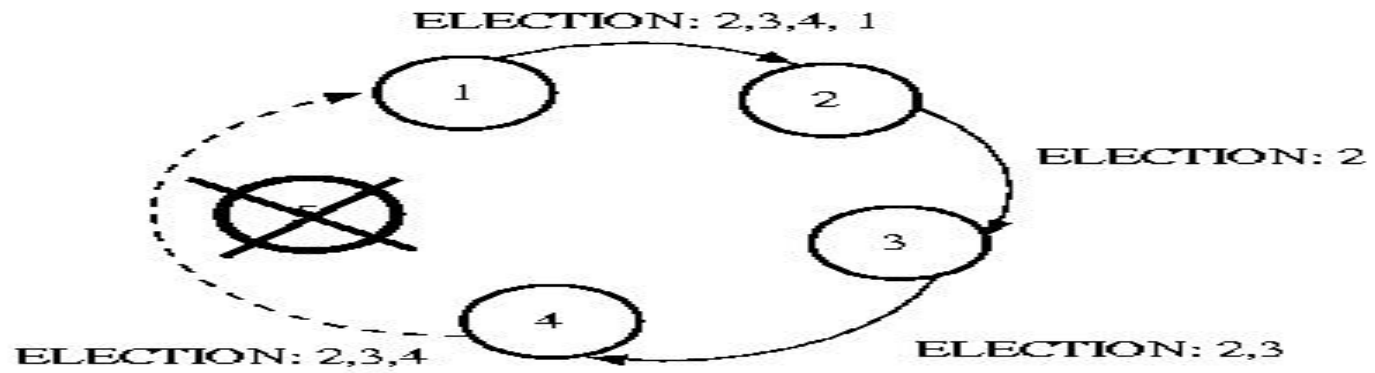


**Fig. 3-12.** The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.



## 2- A Ring algorithm

- We assume the processes is ordered like a ring (i.e. each node knows its successor).
- When a process notices that coordinator is not working, it starts an election by sending message containing its number to its successor.
- If the successor is down, it skips to the next in the ring.
- At each step, the sender adds its process no. to the list in the message.
- When the whole circle is done, and the initiator process receives the message again, this means all are willing to accept the highest number process as a coordinator.
- It then sends a **COORDINATOR** message to all to inform them who became the coordinator.



# Atomic Transactions

Now look at a modern banking application that updates an online data base in place. The customer calls up the bank using a PC with a modem with the intention of withdrawing money from one account and depositing it in another. The operation is performed in two steps:

1. Withdraw(amount, account1).
2. Deposit(amount, account2).

If the telephone connection is broken after the first one but before the second one, the first account will have been debited but the second one will not have been credited. The money vanishes into thin air.

Being able to group these two operations in an atomic transaction would solve the problem. Either both would be completed, or neither would be completed. The key is rolling back to the initial state if the transaction fails to complete. What we really want is a way to rewind the data base as we could the

# Crashes

Client 1:

```
UPDATE Accounts  
SET balance= balance - 500  
WHERE name= 'Fred'
```

```
UPDATE Accounts  
SET balance = balance + 500  
WHERE name= 'Joe'
```

Crash !

What's wrong ?

# Transactions: Definition

- **A transaction** = one or more operations, which reflects a single real-world transition
  - Happens completely or not at all; all-or-nothing
- **Examples**
  - Transfer money between accounts
  - Rent a movie; return a rented movie
  - Purchase a group of products
  - Register for a class (either waitlisted or allocated)
- By using transactions, all previous problems disappear

# Transactions in Applications

START TRANSACTION

[SQL statements]

COMMIT or ROLLBACK (=ABORT)

first SQL query  
starts txn

In ad-hoc SQL: each statement = one transaction

# The Transaction model

- The system is assumed to contain a number of independent processes, each of them can fail at random time due to for example communication link failure.

Transactions have four essential properties. Transactions are:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.


These properties are often referred to by their initial letters, **ACID**.

# The Transaction Model

- Examples of primitives for transactions.

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise





```
BEGIN_TRANSACTION
  reserve WP-JFK;
  reserve JFK-Nairobi;
  reserve Nairobi-Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve WP-JFK;
  reserve JFK-Nairobi;
  Nairobi-Malindi full ⇒ ABORT_TRANSACTION;
```

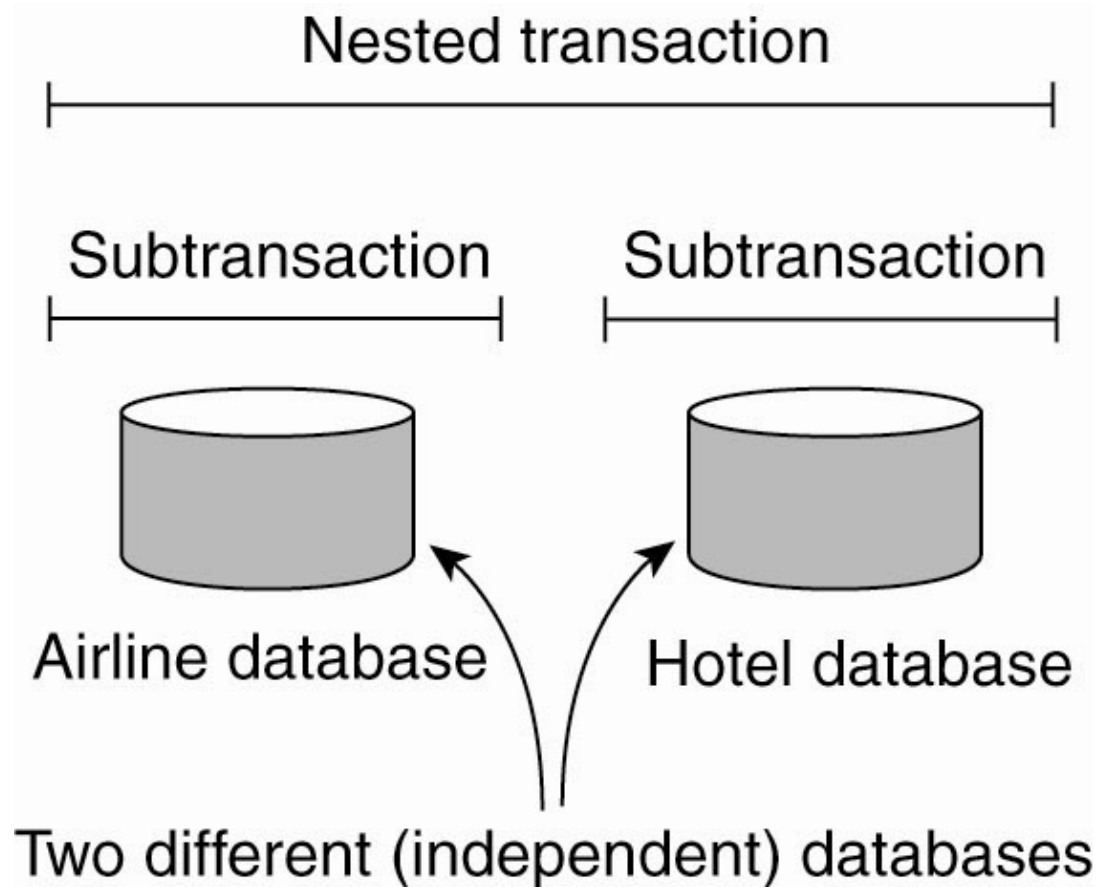
(b)

**Fig. 3-16.** (a) Transaction to reserve three flights commits. (b) Transaction aborts when third flight is unavailable.

# Types of Transactions

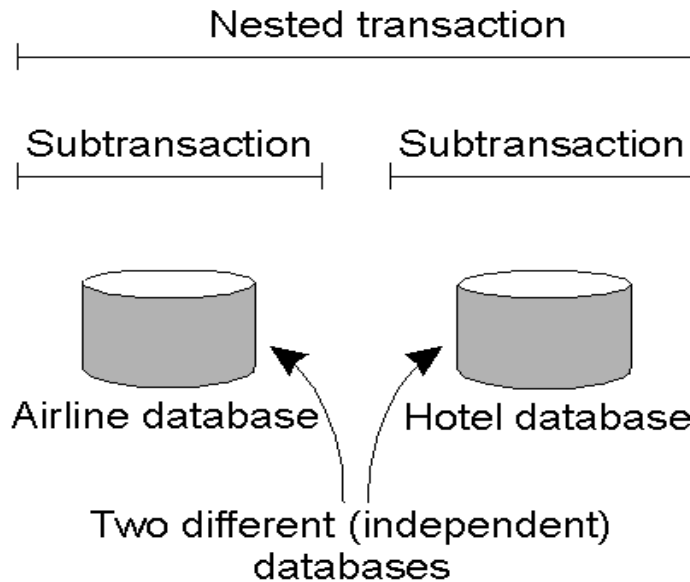
- **Flat Transaction:** this is the model that we have looked at so far.
- **Nested Transaction:** a main, parent transaction spawns child sub-transactions to do the real work. Disadvantage: problems result when a sub-transaction commits and then the parent aborts the main transaction.
- **Distributed Transaction:** this is sub-transactions operating on distributed data stores. Disadvantage: complex mechanisms required to lock the distributed data, as well as commit the entire transaction.

# Nested Transaction

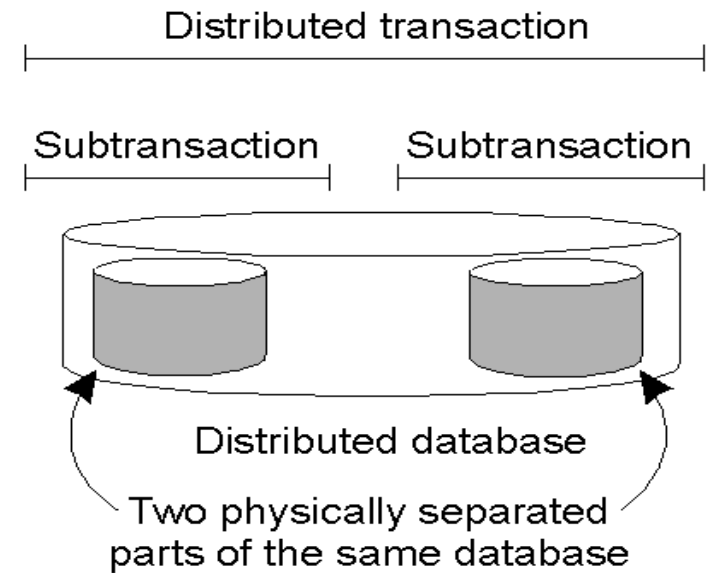


A nested transaction

# Nested vs. Distributed Transactions



(a)



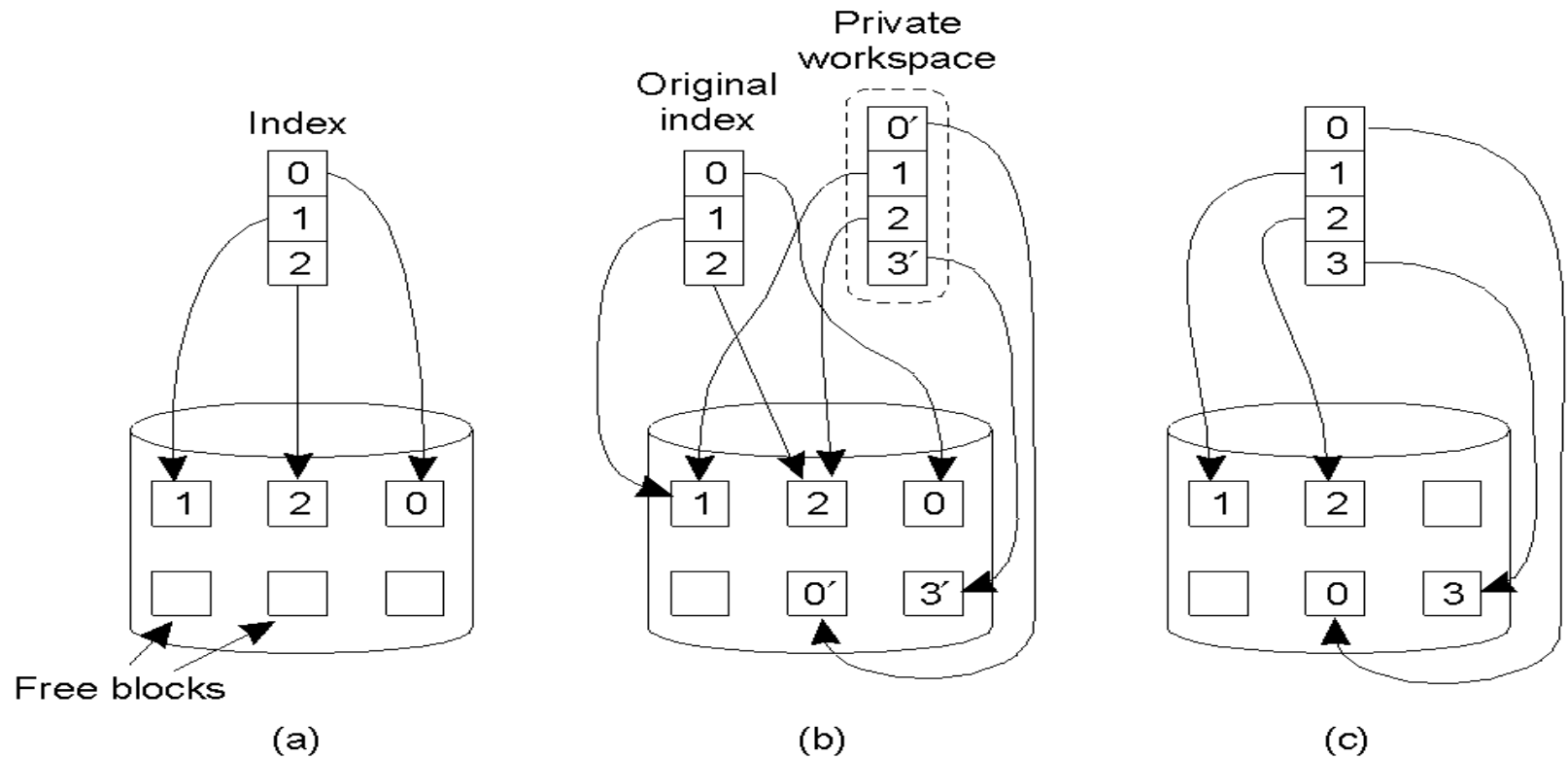
(b)

A distributed transaction – logically a flat, indivisible transaction that operates on distributed data.

# Implementing Transactions by Private workspace

- When a **process** starts a transaction, it is given a **private workspace** containing all the files and objects it will use.
- All the reads and writes are made to this workspace until the transaction commits or aborts.
- When it **commits**, these data are **reflected to the real data**.
- Otherwise, if it **aborts**, all the **workspace is erased without any effect on the real data**
- The **disadvantage** is that it is very costly to copy everything to private workspace specially for large number of processes.
- **Another variation** is that if the process will only read, there is no need for private workspace

# Private Workspace



- a) **The file index and disk blocks for a three-block file**
- b) **The situation after a transaction has modified block 0 and appended block 3**
- c) **After committing**

# Implementing Transactions by writeahead log (intensions list)

- **Files are modified** at each part of the transaction but there is a **log** for each changed data which contains:
- The transaction which made the change – which file and block is being changed – the old and new values being changed.
- If it is aborted, we use the log to return the original values.

```
x = 0;  
y = 0;  
BEGIN_TRANSACTION  
  x = x + 1;  
  y = y + 2;  
  x = y * y;  
END_TRANSACTION
```

(a)

Log

x = 0/1
---------

(b)

Log

x = 0/1
y = 0/2

(c)

Log

x = 0/1
y = 0/2
x = 1/4

(d)

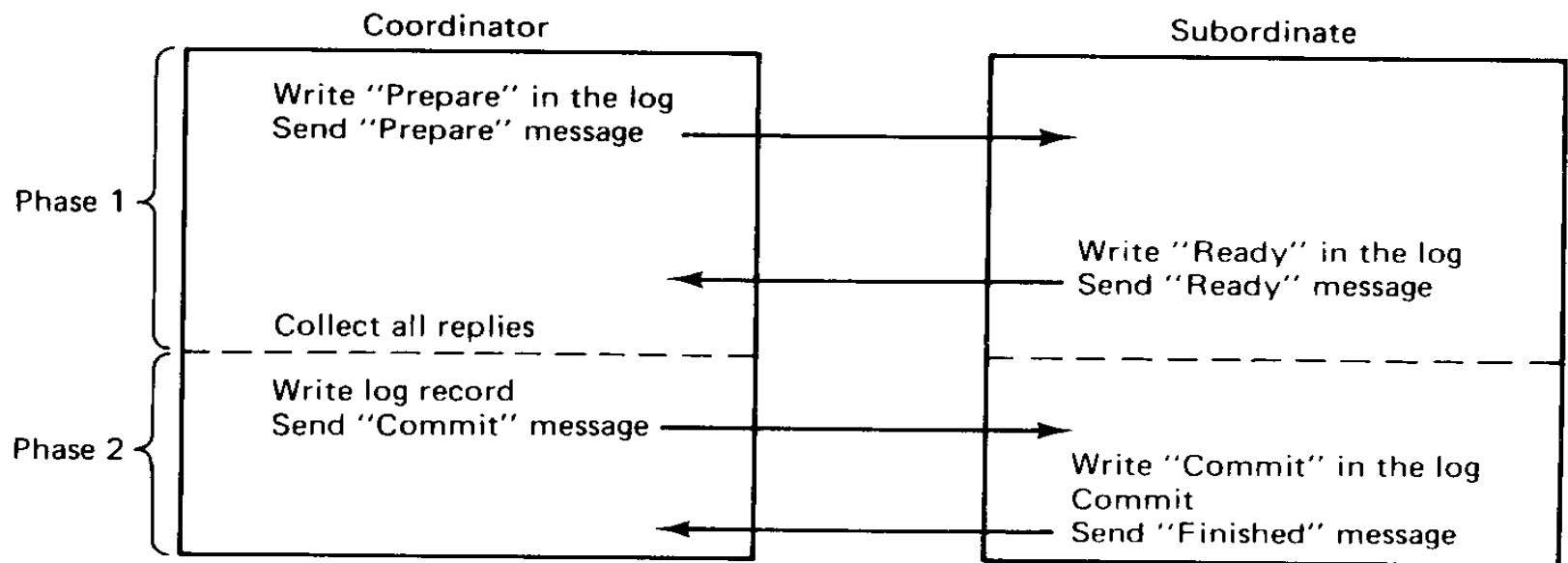
**Fig. 3-19.** (a) A transaction. (b)–(d) The log before each statement is executed.

# Two-phase commit protocol

- It is the most widely used protocol in DS.
- In distributed systems, the commit may require the **cooperation of multiple processes** on different machines.
- One process will be the **coordinator**, which is **executing the transaction** while the other are called **subordinator**.
- **First**, the coordinator process write a log that it is beginning the transaction by starting the commit protocol and sends a message to the subordinators involved in the transaction to prepare to commit.



When a subordinate gets the message it checks to see if it is ready to commit, makes a log entry, and sends back its decision. When the coordinator has received all the responses, it knows whether to commit or abort. If all the processes are prepared to commit, the transaction is committed. If one or more are unable to commit (or do not respond), the transaction is aborted. Either way, the coordinator writes a log entry and then sends a message to each subordinate informing it of the decision. It is this write to the log that actually commits the transaction and makes it go forward no matter what happens afterward.



**Fig. 3-20.** The two-phase commit protocol when it succeeds.

# Concurrency Control

- Multiple concurrent transactions  $T_1, T_2, \dots$
- They read/write common elements  $A_1, A_2, \dots$
- How can we prevent unwanted interference ?

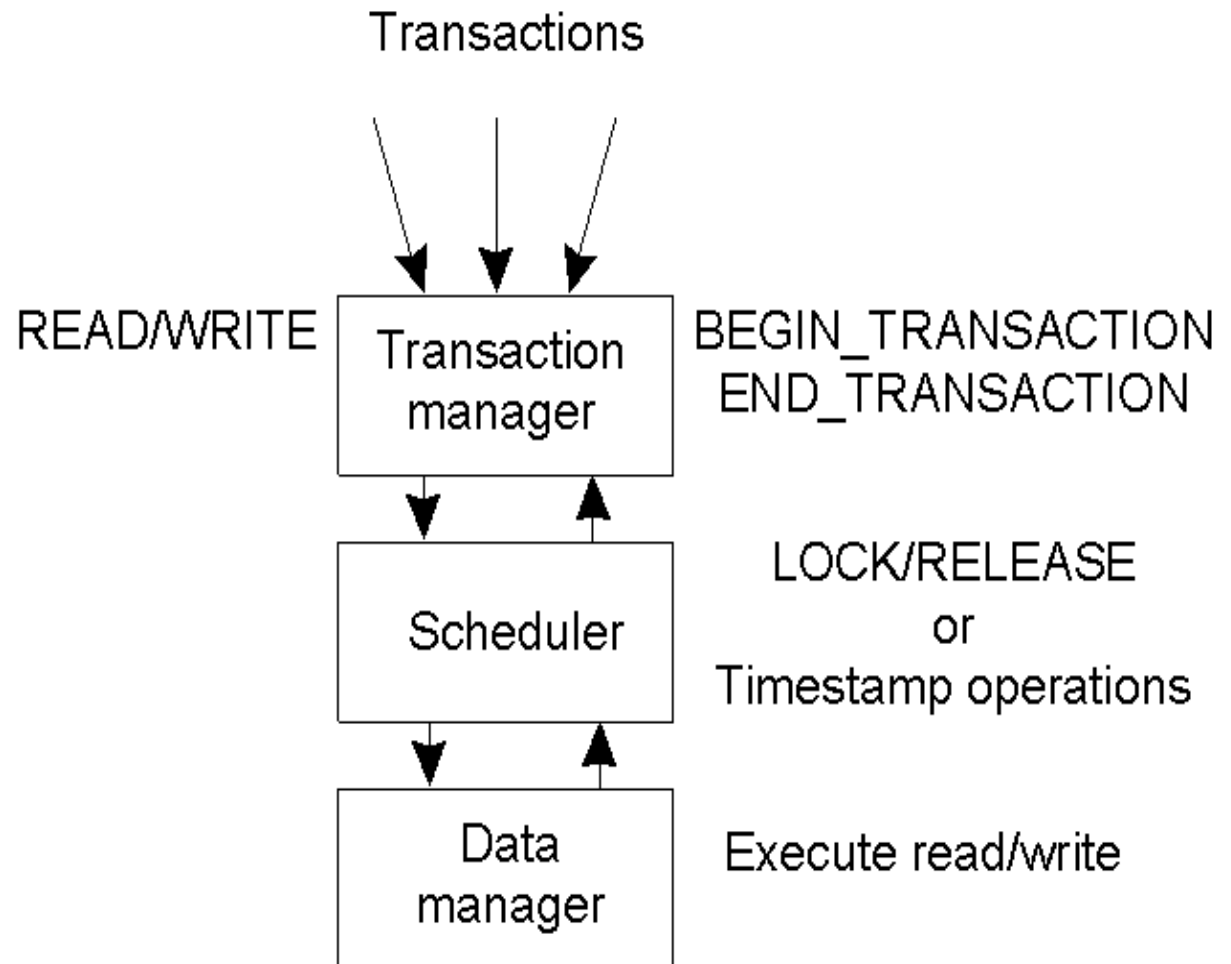
The SCHEDULER is responsible for that

# Schedules

A *schedule* is a sequence of interleaved actions from all transactions

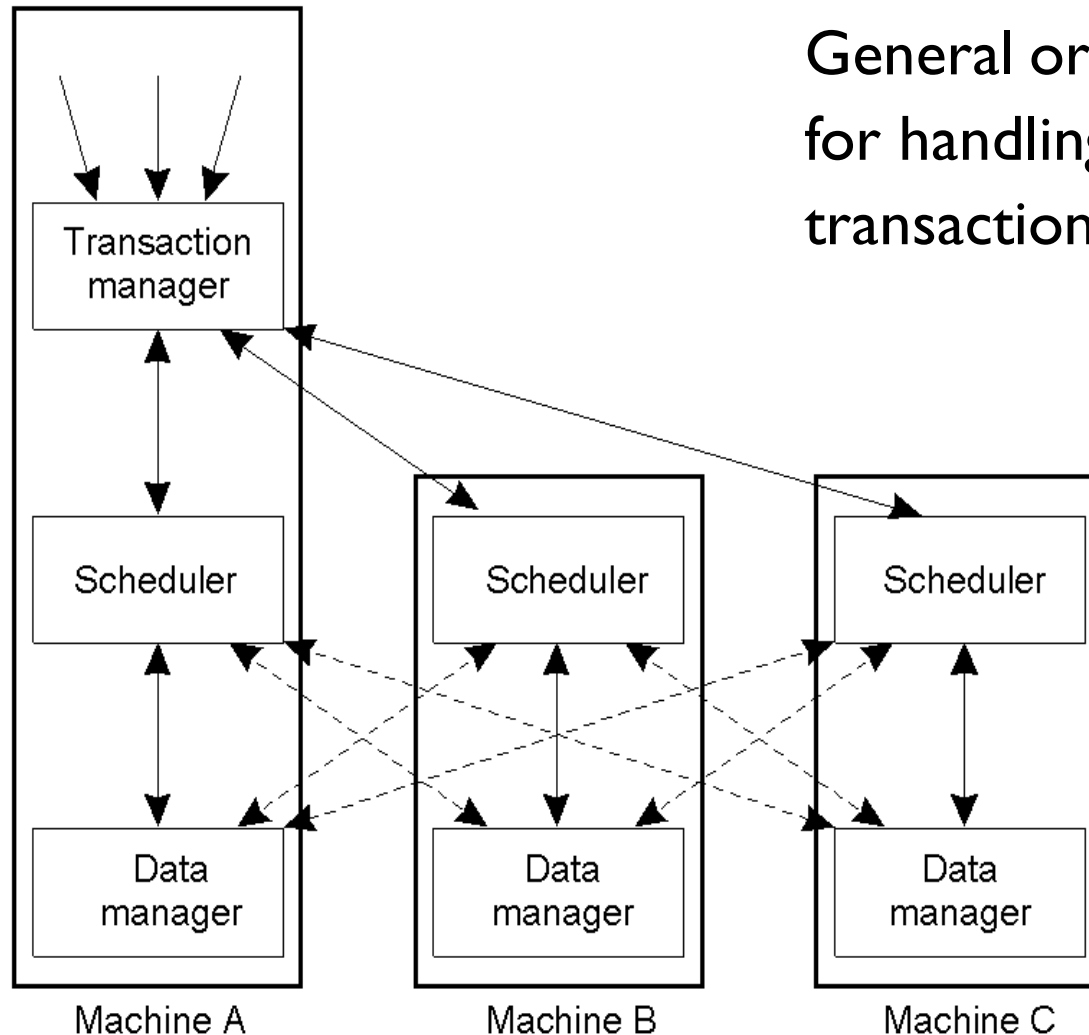
# Concurrency Control

General organization of managers for handling transactions.



# Concurrency Control (2)

General organization of managers for handling distributed transactions.



# Serializability

BEGIN\_TRANSACTION  
x = 0;  
x = x + 1;  
END\_TRANSACTION

(a)

BEGIN\_TRANSACTION  
x = 0;  
x = x + 2;  
END\_TRANSACTION

(b)

BEGIN\_TRANSACTION  
x = 0;  
x = x + 3;  
END\_TRANSACTION

(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

(d)

a) – c) Three transactions  $T_1$ ,  $T_2$ , and  $T_3$

d) Possible schedules

# Achieving concurrency by Locking

The oldest and most widely used concurrency control algorithm is **locking**. In the simplest form, when a process needs to read or write a file (or other object) as part of a transaction, it first locks the file. Locking can be done using a single centralized lock manager, or with a local lock manager on each machine for managing local files. In both cases the lock manager maintains a list of locked files, and rejects all attempts to lock files that are already locked by another process. Since well-behaved processes do not attempt to access a file before it has been locked, setting a lock on a file keeps everyone else away from it and thus ensures that it will not change during the lifetime of the transaction. Locks are normally acquired and released by the transaction system and do not require action by the programmer

# Optimistic concurrency control

- Every transaction assumes there is no others and works on its way until a problem occurs when conflicting with other transaction.
- It checks the status of the data before it begins and when it tries to commit.
- If it was changed by other process while it was working, it aborts else it simply commit.



# Timestamps

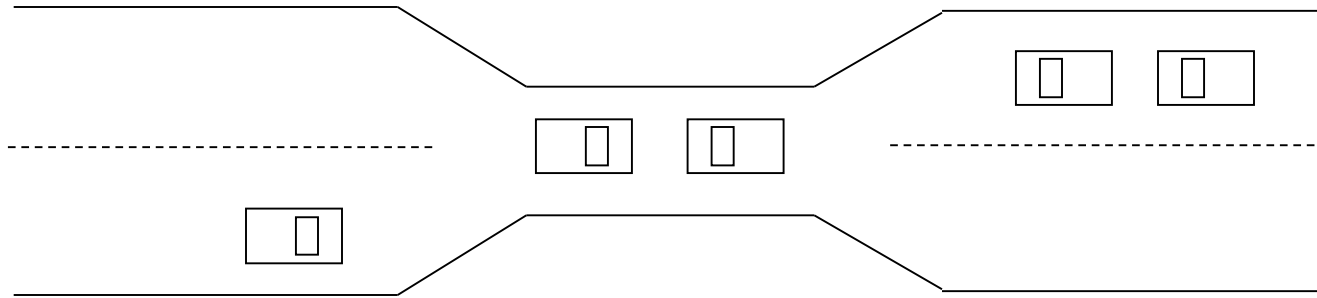
- Each transaction is given a timestamp at the moment it BEGIN transaction
- Using Lamport, we can ensure the timestamp is unique.
- Each file will have a timestamp to know when it was read or written.
- When a transaction tries to access the file, it checks its beginning stamp with the file stamp. It should be higher than it to ensure no one else is using the file.

# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
  - System has 2 disk drives
  - $P_1$  and  $P_2$  each hold one disk drive and each needs another one
- Example
  - semaphores  $A$  and  $B$ , initialized to 1

$P_0$	$P_1$
wait (A);	wait(B)
wait (B);	wait(A)

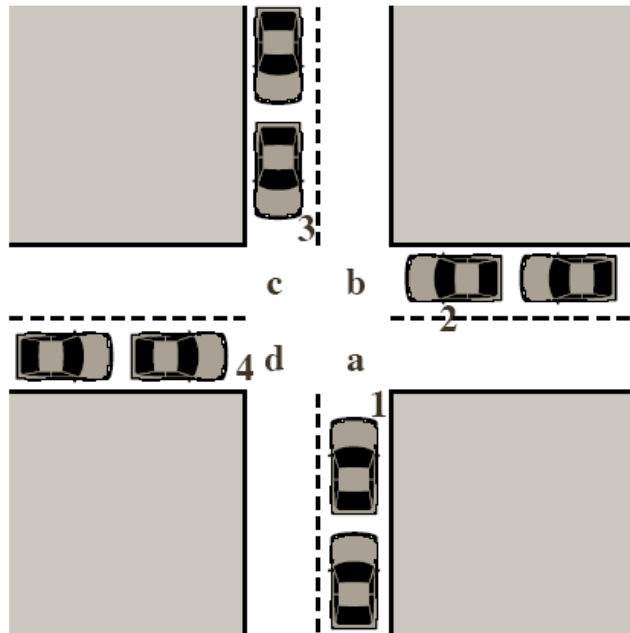
# Bridge Crossing Example



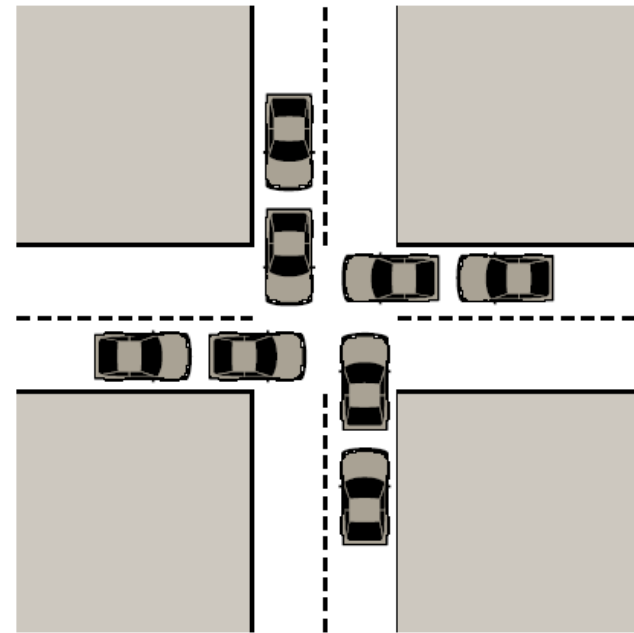
- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up
  - preempt resources and rollback
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- **Note** : Most OSes do not prevent or deal with deadlocks

# Deadlock Principles

- A deadlock is a permanent blocking of a set of threads
  - ✓ a deadlock can happen while threads/processes are competing for system resources or communicating with each other



(a) Deadlock possible



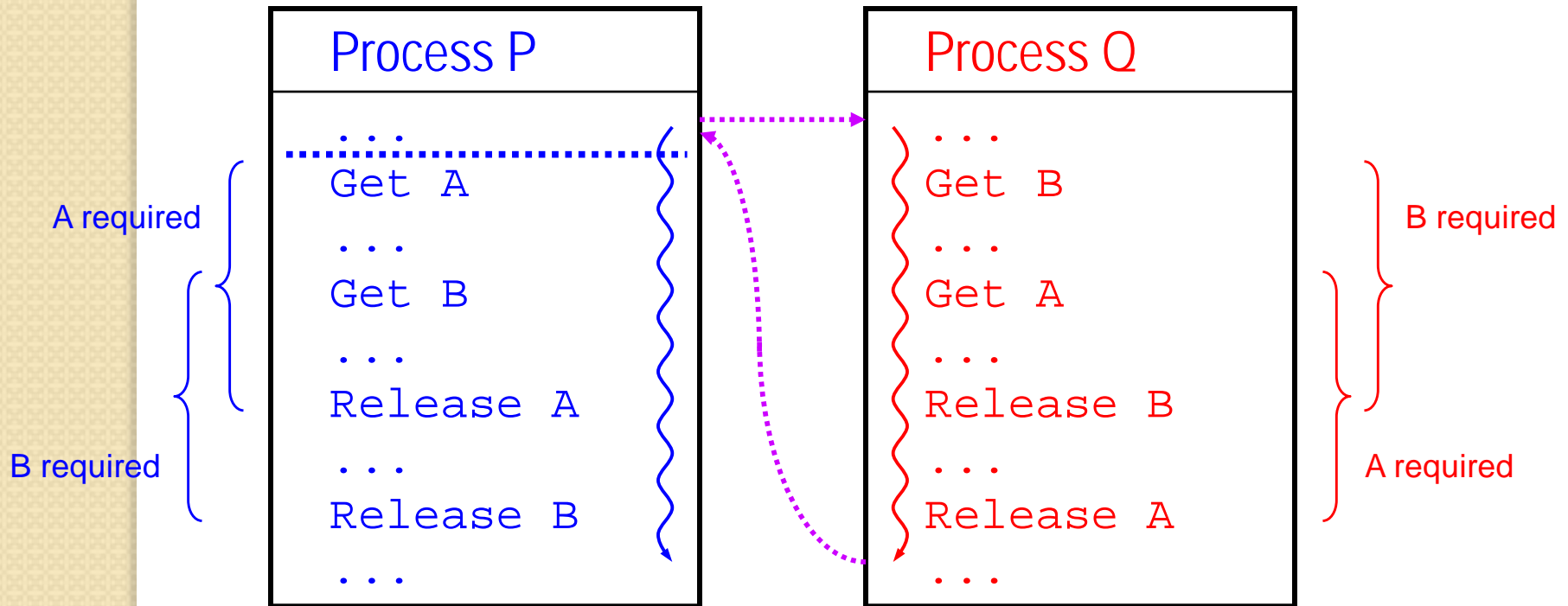
(b) Deadlock

Illustration of a deadlock

# Deadlock Principles

## ➤ Illustration of a deadlock — scheduling path 1 😊

- ✓ Q executes everything before P can ever get A
- ✓ when P is ready, resources A and B are free and P can proceed

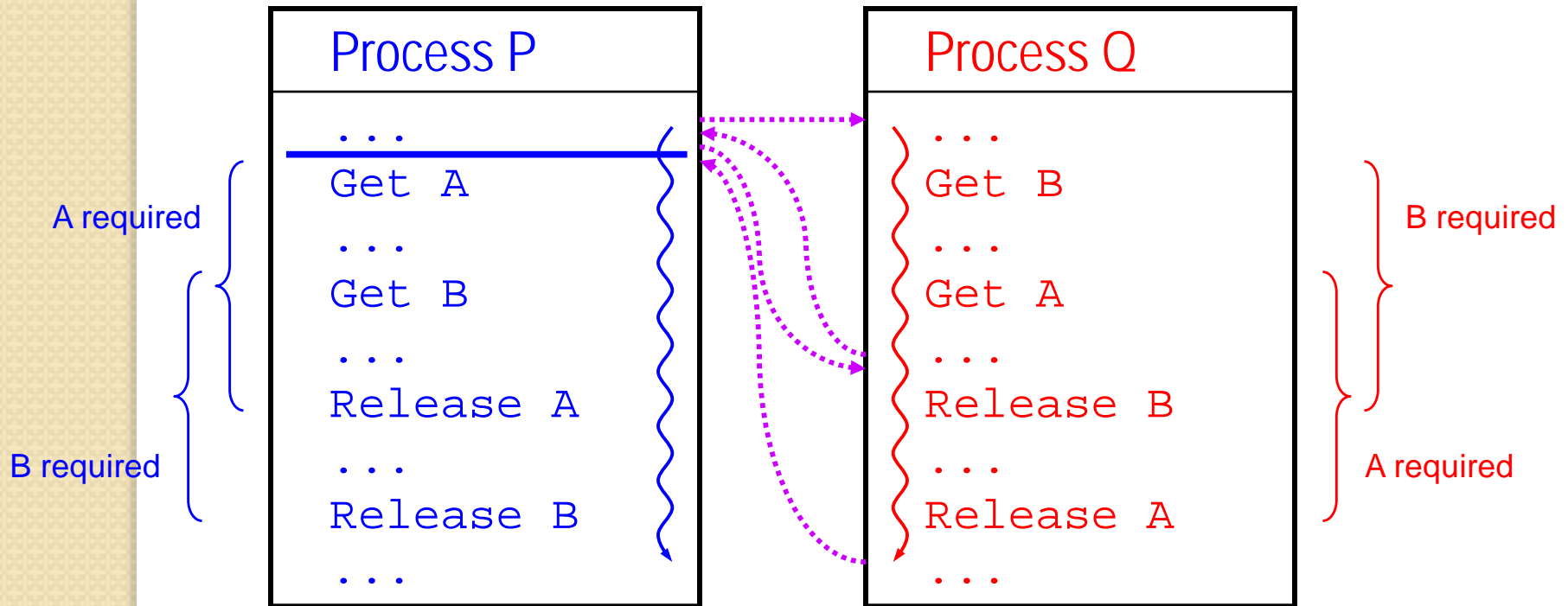


Happy scheduling 1

# Deadlock Principles

## ➤ Illustration of a deadlock — scheduling path 2 😊

- ✓ Q gets B and A, then P is scheduled; P wants A but is blocked by A's mutex; so Q resumes and releases B and A; P can now go

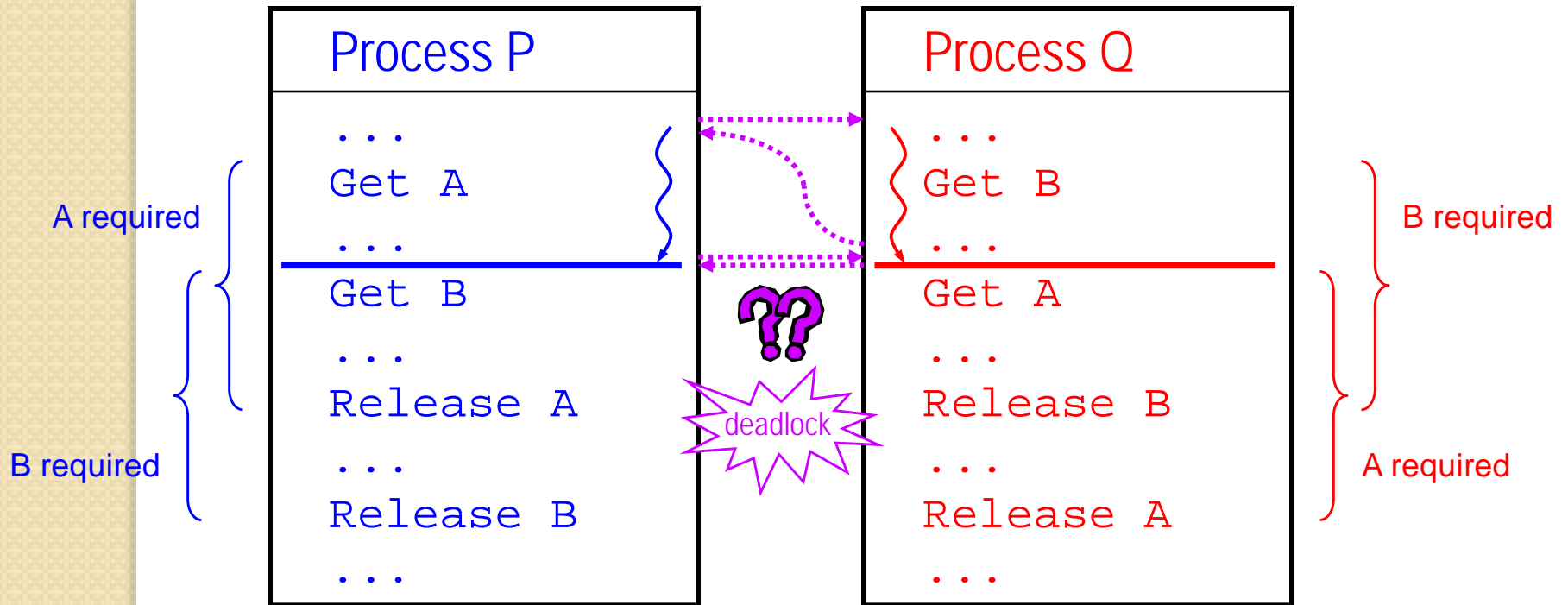


Happy scheduling 2

# Deadlock Principles

## ➤ Illustration of a deadlock — scheduling path 3 😞

- ✓ Q gets only B, then P is scheduled and gets A; now both P and Q are blocked, each waiting for the other to release a resource



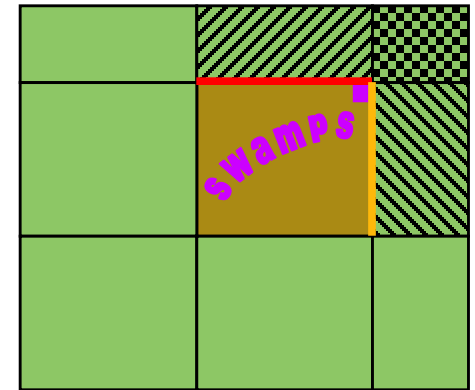
Bad scheduling → **deadlock**

# Deadlock Principles

## ➤ Deadlocks depend on the program and the scheduling

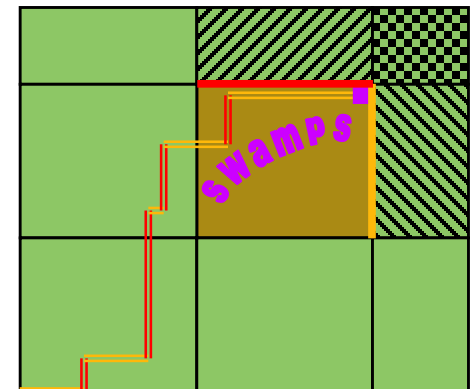
### ✓ program design

- the order of the statements in the code creates the “landscape” of the joint progress diagram
- this landscape may contain gray “swamp” areas leading to **deadlock**



### ✓ scheduling condition

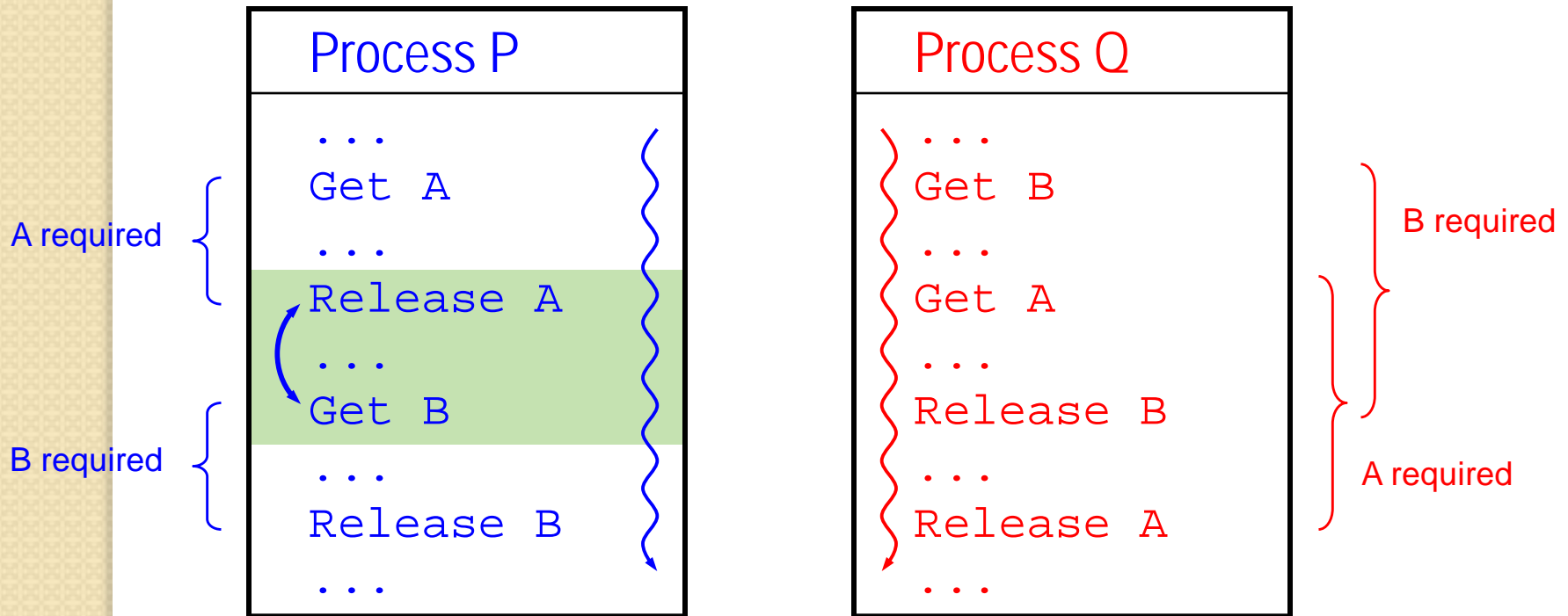
- the interleaved dynamics of multiple executions traces a “path” in this landscape
- this path may sink in the swamps





# Deadlock Principles

- Changing the program changes the landscape
- ✓ here, P releases A before getting B
  - ✓ deadlocks between P and Q are not possible anymore



Competing processes

# System Model

- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

# Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

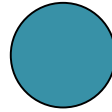
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set  $\{P_0, P_1, \dots, P_0\}$  of waiting processes such that  $P_0$  is waiting for a resource

# Resource-Allocation Graph

A set of vertices  $V$  and a set of edges  $E$ .

- $V$  is partitioned into two types:
  - $P = \{P_1, P_2, \dots, P_n\}$ , the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \dots, R_m\}$ , the set consisting of all resource types in the system
- **request edge** – directed edge  $P_i \rightarrow R_j$
- **assignment edge** – directed edge  $R_j \rightarrow P_i$

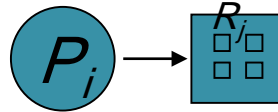
- Process



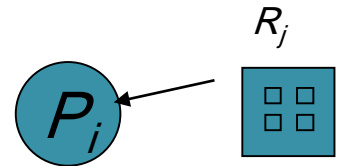
- Resource Type with 4 instances



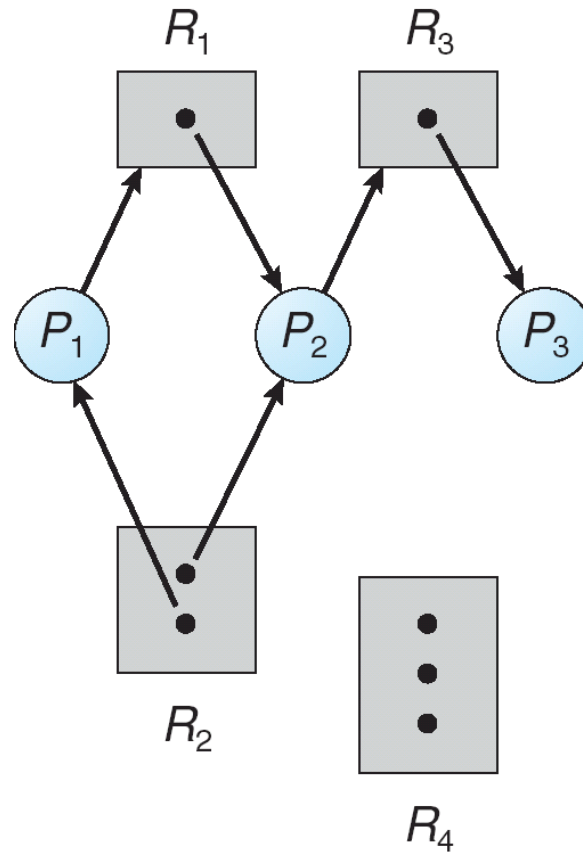
- $P_i$  requests instance of  $R_j$



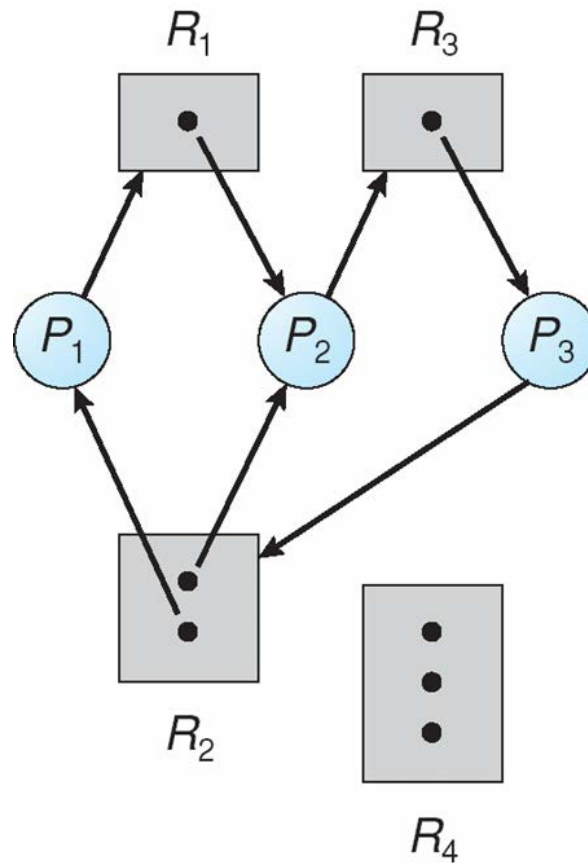
- $P_i$  is holding an instance of  $R_j$



# Example of a Resource Allocation Graph



# Resource Allocation Graph With A Deadlock



# Graph With A Cycle But No Deadlock

